ELSEVIER

Theoretical Computer Science 293 (2003) 391–416

Theoretical Computer Science

www.elsevier.com/locate/tcs

Counting with range concatenation grammars

P. Boullier

INRIA-Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France

Abstract

The class of range concatenation grammars appears to be a convincing challenger as a syntactic base for various tasks, especially in natural language processing. These grammars are powerful, since they strictly contain the class of mildly context-sensitive formalisms, while staying computationally tractable, since their sentences can be parsed in polynomial time. The output of their parsers are structures of polynomial size that can be seen as a generalization of classical context-free shared forest. Moreover, this formalism allows a form of modularity which may lead to the design of libraries of reusable grammatical components. And, finally, it can act as a syntactic backbone upon which decorations from other domains (say feature structures) can be grafted. In this paper we explore the behavior of range concatenation grammars in counting, a domain in which bad reputation of other classical syntactic formalisms is well known. This study leads to some surprising results.

Keywords: Grammar formalisms; Context-sensitive parsing; Shared forests; Parse time complexity; Grammar modularity; Formal power

1. Introduction

The syntactic formalism of range concatenation grammar (RCGs) is a variant of the simple version of literal movement grammars (LMGs), described in [1], and is also related to the framework of LFP developed in [2]. In fact it may be considered to lie halfway between their respective *string* and *integer* versions; RCGs retain from the string version of LMGs or LFPs the notion of concatenation, applying it to ranges (ordered pairs of integers which denote occurrences of substrings in an input string) rather than strings, and from their integer version the ability to handle only (part of) the input string (this later feature is the key to tractability). The definition rules of RCGs, which can be seen as rewriting rules, are called *clauses* and apply to composite objects named *predicates* which are nonterminal symbols with arguments. These arguments

E-mail address: pierre.boullier@inria.fr (P. Boullier).

0304-3975/03/$ - see front matter
PII: S0304-3975(01)00353-X

denote ranges and a predicate (recursively) defines a notion, a property, by specifying the contents of its ranges.

We have shown in [3] that the positive version of RCGs, as simple LMGs or integer indexing LFPs, defines a class of languages called *range concatenation languages* (RCLs) that exactly covers the class *PTIME* of languages recognizable in deterministic polynomial time. Since the rewriting operations of RCGs are not restricted to be linear or non-erasing, some RCLs are not semi-linear. Therefore, RCGs are *not* mildly context-sensitive (MCS) [4] but they are more powerful than, for example, linear context-free rewriting systems (LCFRS) [5] while staying computationally tractable: sentences can be parsed in polynomial time. Within the RCG framework, as in the context-free (CF) case, any sequence of rewriting operations can be represented by a tree, whose root is the object to be derived from. Such a tree abstracts all possible rewriting strategies (i.e., top-down, bottom-up, mixed, etc.) and the parent relation is the only dependence relation. If a tree is rooted at some starting object, and is complete, we have a *parse tree*. For a given sentence, as in the CF case [6], the set of all its parse trees can, in all cases, be packed into a polynomial sized *parse forest*.

Besides its power and efficiency, this formalism has many other attractive properties among which we can mention its closure under both intersection and complement. Since these closure properties are established without changing the component grammars, we conclude that RCGs have a form of modularity which may lead to the design of libraries of reusable grammatical components. And, last, like CF grammars (CFGs), RCGs can act as a syntactic backbone upon which decorations from other domains (e.g., feature structures) can be grafted.

This article is divided in two parts. The first one is dedicated to the definition of RCGs and to a reminder of their main properties. [Classical syntactic formalisms have the reputation to be bad at sums.] In the second part, we explore the behavior of RCGs in counting. This study is performed by means of example languages. Its purpose is not to show that these languages are RCLs, since they can be realized in polynomial time on a Turing machine, but mainly how they can be specified in RCGs and at which (parsing) cost. This last point leads to some surprising results.

2. Range concatenation grammars

This section introduces the notion of RCG and presents some of its properties; more details appear in [7, 8].

2.1. Positive range concatenation grammars

A *positive range concatenation grammar* (PRCG) $G=(N,T,V,P,S)$ is a 5-tuple where N is a finite set of *nonterminal symbols* (also called *predicate names*), T and V are finite, disjoint sets of *terminal symbols* and *variable symbols*, respectively, $S\in N$

is the *axiom*, and P is a finite set of *clauses*

$$\psi_0 \to \psi_1 \dots \psi_j \dots \psi_m$$

where $m \geqslant 0$ and each of $\psi_0, \psi_1, \dots, \psi_m$ is a *predicate* of the form

$$A(\alpha_1, \dots, \alpha_i, \dots, \alpha_p)$$

where $p \geqslant 1$ is its *arity*, $A \in N$ and each of $\alpha_i \in (T \cup V)^*$, $1 \leqslant i \leqslant p$, is an *argument*. In a clause, its left-hand side ψ_0 is a predicate *definition* while in its right-hand side the ψ_j's, $1 \leqslant j \leqslant m$ are predicate *calls*. If the left-hand side of a clause has the form $A(\alpha_1, \dots, \alpha_p)$, we have an A-clause.

Each nonterminal $A \in N$ has a fixed arity whose value is $arity(A)$. By definition $arity(S) = 1$. The *arity* of an A-clause is the arity of A. The *arity* k of a grammar (resulting in a k-PRCG), is the maximum arity of its nonterminals. The *size* of a clause $c = A_0(\dots) \to A_1(\dots) \dots A_j(\dots) \dots A_m(\dots)$ is the integer $|c| = \sum_{j=0}^{m} arity(A_j)$ and the *size* of G is $|G| = \sum_{c \in P} |c|$.

In the sequel, lower case letters such as $a, b, c, \dots$ denote terminal symbols, whereas upper case letters occurring later in the alphabet — such as X, Y, Z — denote variable symbols.

The language defined by a PRCG is based on the notion of *range*. For a given string $w = a_1 \dots a_n \in T^*$, a pair of integers (i, j) such that $0 \leqslant i \leqslant j \leqslant n$ is called a *range*, and is denoted $\langle i..j \rangle_w$, or $\langle i..j \rangle$ when w is understood or when it is of no importance. In $\langle i..j \rangle$ i is its *lower bound*, j is its *upper bound* and $j - i$ is its *size*. If $i = j$, we have an *empty* range. For a given w, the set of all ranges is noted $\mathscr{R}_w$. In fact, $\langle i..j \rangle_w$ denotes the occurrence of the string $a_{i+1} \dots a_j$ in w. Two ranges $\langle i..j \rangle_w$ and $\langle k..l \rangle_w$ can be *concatenated* if and only if the two bounds j and k are equal, the result is the range $\langle i..l \rangle_w$.

In any PRCG, terminal symbols, variables and arguments in a clause denote ranges. The empty argument binds to an empty range. A terminal t binds to the range $\langle j-1..j \rangle_w$ if and only if $w = a_1 \dots a_n$ and $t = a_j$. More generally, the string XY binds to a range if and only if both X and Y denote ranges that can be concatenated: the concatenation on strings matches the concatenation on ranges.

For some $w \in T^*$, we say that $A(\rho_1, \dots, \rho_i, \dots, \rho_p)$ is an *instantiation* of the predicate $A(\alpha_1, \dots, \alpha_i, \dots, \alpha_p)$ if and only if $\rho_i \in \mathscr{R}_w, 1 \leqslant i \leqslant p$ and each symbol (terminal or variable) of $\alpha_1, \dots, \alpha_i, \dots, \alpha_p$ binds to a range in $\mathscr{R}_w$ such that α_i binds to ρ_i, $1 \leqslant i \leqslant p$.

In a clause, several occurrences of the same variable denote the same range, while several occurrences of the same terminal symbol may denote different ranges. If, in a clause, all its predicates are instantiated, we have an *instantiated clause*.

For example, $A(\langle g..h \rangle, \langle i..j \rangle, \langle k..l \rangle) \to B(\langle g+1..h \rangle, \langle i+1..j-1 \rangle, \langle k..l-1 \rangle)$ is an instantiation of the clause $A(aX, bYc, Zd) \to B(X, Y, Z)$ if the input string $a_1 \dots a_n$ is such that $a_{g+1} = a$, $a_{i+1} = b$, $a_j = c$ and $a_l = d$. In this case, the variables X, Y and Z are bound to $\langle g+1..h \rangle$, $\langle i+1..j-1 \rangle$ and $\langle k..l-1 \rangle$, respectively. Often, for a variable X, instead of saying *the range which is bound to* X, we say, *the range* X, or even

instead of *the string whose occurrence is denoted by the range which is bound to X*, we say *the string X*.

For a PRCG $G=(N,T,V,P,S)$ and a string $w\in T^*$, a binary *derive* relation, denoted $\underset{G,w}{\Rightarrow}$, is defined on strings of instantiated predicates. If $\Gamma_1\,\gamma\,\Gamma_2$ is a string of instantiated predicates and if γ is the left-hand side of some instantiated clause $\gamma\to\Gamma$, then we have $\Gamma_1\,\gamma\,\Gamma_2 \underset{G,w}{\Rightarrow} \Gamma_1\,\Gamma\,\Gamma_2$.

For a string $w\in T^*$ of length n, the range $\langle 0..n\rangle_w$ is called *initial range*. The string w is a sentence if and only if the empty string (of instantiated predicates) can be derived from $S(\langle 0..n\rangle_w)$ (i.e., $S(\langle 0..n\rangle_w) \overset{+}{\underset{G,w}{\Rightarrow}} \varepsilon$, where $\overset{+}{\underset{G,w}{\Rightarrow}}$ is the transitive closure of $\underset{G,w}{\Rightarrow}$). Any sequence such as $\Gamma \underset{G,w}{\Rightarrow} \cdots \underset{G,w}{\Rightarrow} \Gamma'$ is a *derivation* while the sequence $S(\langle 0..n\rangle_w) \underset{G,w}{\Rightarrow} \cdots \underset{G,w}{\Rightarrow} \varepsilon$ is a *complete derivation*. The language $\mathscr{L}(G)$ defined by a PRCG G, is the set of all its sentences. More generally, for each $A\in N$, we can define $\mathscr{L}(A)$, the language of A.

The arguments of a given predicate may bind to discontinuous or even overlapping ranges. Fundamentally, a nonterminal symbol A defines a notion (property, structure, dependency, etc.) between its arguments, whose associated ranges can be arbitrarily scattered over the input string. PRCGs are, therefore, well suited to describe long distance dependencies. Overlapping ranges arise as a consequence of the non-linearity of the formalism. For example, the same variable (denoting the same range) may occur in different arguments in the right-hand side of some clause, expressing different views (properties) of the same portion of the input string.

Note that the order of right-hand side predicates in a clause is of no importance (in fact, the right-hand side of a clause is a set of predicate calls rather than a list).

As an example of a PRCG, the following set of clauses describes the 3-copy language $\{www \mid w\in\{a,b\}^*\}$ which is not a CF language (CFL) and even lies beyond the formal power of tree adjoining grammars (TAGs) (an introduction to TAG can be found in [9]).

$$
\begin{aligned}
S(XYZ) &\to A(X,Y,Z)\\
A(aX,aY,aZ) &\to A(X,Y,Z)\\
A(bX,bY,bZ) &\to A(X,Y,Z)\\
A(\varepsilon,\varepsilon,\varepsilon) &\to \varepsilon
\end{aligned}
$$

Remark that the set of clauses

$$
\begin{aligned}
S(XXX) &\to A(X)\\
A(aX) &\to A(X)\\
A(bX) &\to A(X)\\
A(\varepsilon) &\to \varepsilon
\end{aligned}
$$

does *not* define the 3-copy language. In the first clause all the occurrences of the variable X must bind to the same range, thus, the argument XXX, which bind to the

concatenation of three identical ranges, can be instantiated if and only if X binds to an empty range. It follows that the only sentence of this language is the empty string.

2.2. Negative range concatenation grammars

A *negative range concatenation grammar* (NRCG) $G=(N,T,V,P,S)$ is a 5-tuple, like a PRCG, except that some predicates occurring in the right-hand side have the form $\overline{A(\alpha_1,\ldots,\alpha_p)}$.

A predicate call of the form $\overline{A(\alpha_1,\ldots,\alpha_p)}$ is said to be a *negative predicate call.*

A *range concatenation grammar* (RCG) is a PRCG or an NRCG.

The PRCG (resp. NRCG) term is used to emphasize the absence (resp. presence) of negative predicate calls.

In an NRCG, the intended meaning of a negative predicate call is to define the complement language (with respect to T^*) of its positive counterpart: an instantiated negative predicate succeeds if and only if its positive counterpart (always) fails. This definition is based on a "negation by failure" rule (see [8] for a more precise discussion). However, in order to avoid inconsistencies occurring when an instantiated predicate can derive its own negative counterpart, we prohibit *inconsistent* derivations exhibiting this possibility. More precisely, we reject *inconsistent* grammars in which inconsistent derivations may occur. Informally, in inconsistent grammars, some sublanguage is defined as being its own complement as by means of $A(X)\rightarrow\overline{A(X)}$. Thus, in the sequel, we assume that any NRCG is *consistent*. However, the decidability of its consistence is, for an NRCG, an open problem.

2.3. Subclasses

Partly borrowing from [1], we say that a clause is:

- *bottom-up erasing* (resp. *top-down erasing*) if there is at least one variable occurring in its right-hand side (resp. left-hand side) which does not appear in its left-hand side (resp. right-hand side);
- *erasing* if there exists a variable appearing only in its left-hand side or only in its right-hand side;
- *linear* if none of its variables occurs twice in its left-hand side or twice in its right-hand side;

These definitions extend naturally from clause to set of clauses (i.e., to grammars).

We will assume that RCGs are not bottom-up erasing. In fact we assume a slightly stronger condition which is: in any clause, each right-hand side argument must contain at least one variable and must be a substring of some left-hand side argument. This constraint is (abusively) called bottom-up non-erasing (*bune*) condition. If the left-hand side predicate of a clause is instantiated, and if all the symbols (elements of $(T\cup V)$) of its left-hand side arguments are instantiated, the bune condition implies that all right-hand side arguments are uniquely instantiated. An argument of a predicate A is termed as *universal* if and only if, for each instantiated A-clause, all the other instantiated arguments are subranges of that universal instantiated argument.

Of course, an argument which is always bound to the initial range is universal. The bune condition does not restrain the class of grammars since any non-bune RCG $G=(N,T,V,P,S)$ can be transformed into an equivalent bune RCG $G_1=(N_1,T_1,V_1,P_1,S_1)$ as follows.

Let q be the maximum number of non-bune arguments in a clause of G. Except for S_1, each nonterminal symbol in N_1 is a nonterminal symbol in N, but its arity is increased by q. In fact, except for the axiom, each predicate in P_1 is equipped with q supplementary universal arguments. The S_1-clause is defined by

$$S_1(X) \to S(X,\underbrace{X,\ldots,X}_{q})$$

Each clause in P is transformed into one clause in P_1 as follows. For simplicity reasons, we only consider here the transformation implied by the non-bune single argument α in:

$$A(\underbrace{\ldots}_{p}) \to \ldots B(\underbrace{\ldots,\alpha,\ldots}_{r})\ldots \in P$$

Assuming that α is processed with the help of the left-hand side rightmost universal argument, two cases can occur.

If $\alpha \in T^*$, the above clause is changed into

$$A(\underbrace{\ldots}_{p},\underbrace{\ldots,W_1\alpha E W_2}_{q}) \to \ldots empty(E)\ B(\underbrace{\ldots,\alpha E,\ldots}_{r},\underbrace{\ldots,W_1\alpha E W_2}_{q})\ldots$$

while, if $\alpha \notin T^*$, it is changed into

$$A(\underbrace{\ldots}_{p},\underbrace{\ldots,W_1\alpha W_2}_{q}) \to \ldots B(\underbrace{\ldots,\alpha,\ldots}_{r},\underbrace{\ldots,W_1\alpha W_2}_{q})\ldots$$

where W_1, E and W_2 are brand new variables, and the unary nonterminal *empty* is defined by

$$empty(\varepsilon) \to \varepsilon$$

2.4. Parse time complexity

In [8], we presented a parsing algorithm which, for an RCG G and an input string of length n, produces its parse forest in time linear with $|G|$ and polynomial with n. The maximum exponent value of this polynomial is called *degree* of G. This degree is the maximum number of free (independent) bounds in a clause.

If $x=a_1\ldots a_p$ is a string on some alphabet, each integer k, $0\leqslant k\leqslant p=|x|$ is called a *position* or a *bound* of x. If k is a position in x, $k>0$, $x(k)$ denotes the symbol a_k. By definition, the position 0 denotes the beginning of the string and we can write $x(0)=\varepsilon$.

When a predicate $\psi = A(\alpha_1, \ldots, \alpha_m)$, $m = arity(A)$ is instantiated in $\gamma = A(\rho_1, \ldots, \rho_m)$, $\rho_j \in \mathscr{R}_w$, $1 \leqslant j \leqslant m$ for some $w \in T^*$, we can compute, for each position of each argument α_j, a mapping to an *input index* which is a position in w. An *item*, associated to an instantiation γ of ψ, is a tuple of the form $(A, i_1^0, \ldots, i_1^{|\alpha_1|}, i_2^0, \ldots, i_m^{|\alpha_m|})$ in which the first component is a nonterminal and the i's are all input indexes: i_j^k is the input index associated to the kth position in the jth argument α_j of ψ. Of course, in this case, we have $\rho_j = \langle i_j^0..i_j^{|\alpha_j|} \rangle_w$, $1 \leqslant j \leqslant m$.

In fact, the main purpose of our parsing algorithm is to build, for each clause c, from any instantiation $\gamma = A(\rho_1, \ldots, \rho_m)$ of its left-hand side predicate $\psi = A(\alpha_1, \ldots, \alpha_m)$, the set of its items. Then, for each item in this item set, the bune condition guarantees a unique instantiation of c.

Thus, our parsing algorithm, which is in spirit very close to a classical recursive descent parser, is organized around a (recursive) boolean function named *prdct* which takes as argument an instantiated predicate, say $A(\rho_1, \ldots, \rho_m)$. As mentioned above, for each A-clause c, this function computes successively the corresponding set of items. For each item in this set, the instantiated predicates in the right-hand side of the corresponding instantiation of c are successively used as calling arguments of the function *prdct*. Moreover, a tabulation mechanism allows to not re-execute an already executed instantiated predicate (or clause) and allows to return its value.

If we assume that this tabulation mechanism, as it is usual, takes a unit time, we see that the time taken by this parsing algorithm is proportional to the number of computed items. Thus, throughout this article, we take this number of items as a measure of parse time complexity. These complexities are expressed using Bachmann's $\mathcal{O}$-notation (see [10]): $\mathcal{O}(f(n))$ stands for a number whose absolute value is at most a constant times $|f(n)|$ (abusing this notation, we use $\mathcal{O}(1)$ to denote a constant).

For an input string w of length n, the initial call to the function *prdct* is performed with $S(\langle 0..n \rangle_w)$ as argument. In order to do that, we must assume that w is already known (i.e., the input is off-line) and appears as a global parameter in the algorithm. Thus, the (linear) time taken by the process which reads and stores the input string on the (random access) memory is *not* part of our complexity measure. The relevance of this harmless remark will appear later on in connection with sublinear parse times. Moreover, we assume that this reading phase checks that w is a string in T^*.

For a language L, defined by a bune RCG G, if r is the maximum number of indexes in an item, the previous study shows that, for an input string of length n (not necessarily a sentence), the time complexity of the parser for L is $\mathcal{O}(n^r)$. However, this upper bound may sometimes be improved by noting that, in an item, the input indexes are not necessarily independent of each other. This is trivially the case when two consecutive bounds (positions) in an argument surround a terminal symbol. In that case the difference between the corresponding input indexes in each associated item is always one. This is also the case when the size of some variable is statically known. This knowledge can be achieved by a grammar analysis or directly by the predefined predicates of the next Section.

For a bune k-RCG G, the number of free bounds is less than or equal to $d = k + \max_{c_j \in P} v_j$, where c_j is the jth clause in P, and v_j is the number of different variables in the left-hand side predicate of c_j. Nonetheless, for any RCG G, the degree d is the number of its free bounds and the parse time of an input string of length n takes $\mathcal{O}(|G|n^d)$.

2.4.1. Predefined predicates

In order to make the design of RCGs easier, we have predefined a certain number of nonterminal symbols among which we mention here *len*, *eqlen* and *eq*:

len(l, α): checks that the size of the range denoted by the argument α is the integer l;

eqlen(α_1, α_2): checks that the sizes of the ranges denoted by the arguments α_1 and α_2 are equal;

eq(α_1, α_2): checks that the substrings of the input text selected by the ranges denoted by the arguments α_1 and α_2 are equal.

It must be noted that these predefined predicates do not increase the formal power of RCGs insofar as each of them can be defined by a pure RCG. For example, the predicate call *len*$(1, X)$ is equivalent to $len_1(X)$, if the nonterminal len_1 is defined by a clause schema such as $len_1(t) \rightarrow \varepsilon$ over all terminals $t \in T$.

Their introduction also is justified by the fact that they are more efficiently implemented than their RCG defined counterparts and, more significantly that they convey static information which can be used to decrease the number of free bounds in the clause in which they occur, and can thus lead to an improved parse time complexity.

For example, the predicate call *len*$(3, X)$, indicates that the size of (the range denoted by) X is 3; this means that the positions in the arguments before and after any occurrence of X are not both free. In an analogous manner, *eqlen*(X, Y) or *eq*(X, Y), implies that among the four lower and upper positions of X and Y, at most three of them are free.

2.5. Closure properties and modularity

In [8] we have shown that RCLs are closed under union, concatenation, Kleene iteration, intersection and complement; the proofs are shown again below.

Let $G_1 = (N_1, T_1, V_1, P_1, S_2)$ and $G_2 = (N_2, T_2, V_2, P_2, S_2)$ be two RCGs defining the languages L_1 and L_2, respectively. Without loss of generality, we assume that $N_1 \cap N_2 = \emptyset$. Let S be a unary nonterminal not in $N_1 \cup N_2$. Consider two RCGs $G' = (N_1 \cup \{S\}, T_1, V_1 \cup \{X, Y\}, P_1 \cup P', S)$ and $G'' = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, V_1 \cup V_2 \cup \{X, Y\}, P_1 \cup P_2 \cup P'', S)$ defining the languages L' and L'', respectively. By careful definition of the additional sets of clauses P' and P'', we can get $L'' = L_1 \cup L_2$, $L'' = L_1 L_2$ or $L'' = L_1 \cap L_2$ and $L' = L_1^*$ or $L' = \overline{L_1}$.

Union: $P'' = \{S(X) \rightarrow S_1(X), S(X) \rightarrow S_2(X)\}$
Concatenation: $P'' = \{S(XY) \rightarrow S_1(X)\ S_2(Y)\}$
Intersection: $P'' = \{S(X) \rightarrow S_1(X)\ S_2(X)\}$

Kleene iteration: $P' = \{S(\varepsilon) \rightarrow \varepsilon, S(XY) \rightarrow S_1(X)\ S(Y)\}$
Complement: $P' = \{S(X) \rightarrow \overline{S_1(X)}\}$

In [7], we have shown that the emptiness problem for RCLs is undecidable and that RCLs are not closed under homomorphism. In fact, we have shown that a polynomial parse time formalism that extends CFGs cannot be closed both under homomorphism and intersection and we advocate that, for an NL description formalism, it is worth being closed under intersection rather than under homomorphism. This is specially true when this closure property is reached without changing the component grammars.

Let G_1 and G_2 be two grammars in some formalism $\mathscr{F}$, their sets of rewriting rules are P_1 and P_2 and they define the languages L_1 and L_2, respectively. We assume that P_1 and P_2 are each independent of the other and thus cannot interfere in any way. We say that $\mathscr{F}$ is *modular* with respect to some closure operation f if the language $L = f(L_1, L_2)$ can be defined by a grammar G in $\mathscr{F}$ whose set of rules P is such that $P_1 \cup P_2 \subseteq P$. The idea behind this notion of subgrammar is to preserve the structures (parse trees for G_1 and G_2) built by the component grammars. In that sense, we can say that CFGs are modular with respect to union operation since, on the one hand, CFLs have the formal property to be closed under union and, on the other hand, this union is described without changing the component grammars G_1 and G_2 (we simply have to add the two rules $S \rightarrow S_1$ and $S \rightarrow S_2$). Conversely, CFGs are not modular with respect to intersection or complement since we know that CFLs are not closed under intersection or complement. If we now consider regular languages, we know that they are closed under intersection and complement; however we cannot say that they are modular with respect to these properties, since the structure of the component grammar is not preserved in any sense. For example, let us take a regular CFG G, defining the language L, we know that it is possible to construct a regular CFG whose language is $\bar{L}$, but its parse trees are not related to the parse trees of G.

Following our definition, we see that RCLs are modular with respect to union, concatenation, Kleene iteration, intersection and complement. We would like to highlight why it is of a considerable benefit for a formalism to be modular with respect to intersection and complement.

Assuming that we have two grammars G_1 and G_2, modularity with respect to intersection allows us to directly define a language whose set of sentences is $\mathscr{L}(G_1) \cap \mathscr{L}(G_2)$, without changing neither G_1 nor G_2.

Modularity with respect to complement (or difference) allows us to handle $\bar{L}$, the complement of some language L, as easily as L.

If modularity with respect to both intersection and complement holds, we can, for example, easily model the paradigm "general rule with exceptions". Assume that we have some language $\mathscr{L}$ which is almost identical to $\mathscr{R}$, except for the subset $\mathscr{S}$, for which its strings must be replaced by the strings of $\mathscr{E}$ (i.e., $\mathscr{L} = \mathscr{R} \cap \bar{\mathscr{S}} \cup \mathscr{E}$). Within the RCG formalism, to define $\mathscr{L}$, if we assume that the unary predicates R, F and E, respectively, define the languages $\mathscr{R}$, $\mathscr{S}$ and $\mathscr{E}$, we simply have to add the two

L-clauses

$$L(X) \rightarrow R(X)\ \overline{F(X)}$$

$$L(X) \rightarrow E(X)$$

3. Counting with RCGs

Usually, we say that a formalism can *count* up to n if it can generate for some n the language $\{a_1^k \ldots a_n^k \mid k \geqslant 0\}$. For example CFGs are able to *count* up to 2, TAGs can *count* up to 4, whereas indexed grammars (IGs) [11] can *count* to an arbitrary fixed number n. Though RCGs can also *count* to an arbitrary fixed number n, this is not this definition of counting we are talking about in this article. For us, the counting process refers much more to elementary arithmetic operations (i.e., addition, subtraction, etc.) that are performed on specific devices called counters and which can be used in any RCG to define some properties of the corresponding language. These counters are not defined by some supplementary mechanism: the grammar designer can decide that some arguments of some nonterminal symbols are counters. Therefore, from the RCG formalism point of view, counters do not play any particular role: they are merely arguments that, like any other argument, bind to ranges. Thus, eventually, counters values are coded by ranges (i.e., parts of the input string). This means that, in some cases, the input text plays two roles: as usual the role of sentence, while some part of it can also play the role of (coded) counter values. This remark implies that only nonlinear formalisms, as RCGs, can define this notion of counters.

As already mentioned, the purpose of this section is not only to show that some kind of counting can be performed with RCGs but also to show how they can be performed. Another goal is to show that this counting mechanism can be efficiently (with respect to the corresponding parse time complexities) specified. This last point would result in RCGs that are not as elegant as they could have been. In the sequel, in many examples, we use RCGs in an imperative way talking for example of *initialization* clause. This (usually top-down) vision adopted for explanation purposes must not occult the fact that the RCG formalism is declarative. For example when we define the nonterminal $[x*y=z]$ for the multiply operation, the corresponding clauses define as well the divide, square and square root operations.

3.1. Counters

In order to perform counting, we must have *counters*. This part can be played by predicate arguments: the grammar designer decides that in a certain predicate, some specific arguments are counters. Counters, as all other arguments, are bound to ranges and the simplest way to assign a value to a counter is to take the size of its associated range. Doing so, the immediate consequence is that counters are non-negative integers whose upper limit is bounded by the length n of the input string. We shall see in Section 3.8 how this constraint can be relaxed, but let us first assume that a counter

is a single dedicated argument. Since the arity of nonterminals is fixed and the set N is finite, for a given grammar, the number of its counters is bounded.

With this observation in mind, we see that the same value can be represented by different ranges. For example, two ranges $\rho_1 = \langle i_1..j_1 \rangle$ and $\rho_2 = \langle i_2..j_2 \rangle$ can be different though their sizes $|\rho_1| = j_1 - i_1$ and $|\rho_2| = j_2 - i_2$ are equal. We can get round this (somehow unpleasant) property by defining a *normal form* for counters in which the lower bound (or the upper bound) of the associated range is always a constant, say i. Then we say that the counter is in normal form at i, or shortly in normal form if the origin is of no importance. Usually, we take $i = 0$, the lower bound of any initial range. Thus, the equality of counters in normal form becomes the identity of argument strings.

In the sequel, in order to simplify the notations, variables whose length h is statically known will all be denoted by an upper case T together with h as superscript (i.e., T^h). If in a clause different variables have the same length, we provide T with indices.

For example the clause

$$A(XY, ZW, E) \rightarrow len(2, X)\ len(2, Z)\ len(0, E)\ B(\ldots)\ldots$$

can be written as

$$A(T_1^2 Y, T_2^2 W, T^0) \rightarrow B(\ldots)\ldots$$

without any further explanation.

The *assignment* operation of (the value of) a counter to another counter can be performed by duplication of the corresponding argument as in

$$\ldots \rightarrow A(X, \ldots)B(X, \ldots)\ldots$$

where the first arguments of A and B are counters. This operation is allowed because of the non-linearity of the RCG formalism.

We now show how a counter in normal form at i can be created and initialized with the value of another counter (not in normal form). In order to do that, we assume the presence of a universal argument (see Section 2.3) whose lower bound is i. Thus, if a normal form is required for the first argument of B, the value of which must be the counter located in the first argument of A, assuming that the last argument of A is universal with lower bound i, we can write something like

$$A(X, \ldots, YW) \rightarrow \ldots eqlen(X, Y)B(Y, \ldots)\ldots$$

The *predecessor* operation can be expressed by something like

$$A(XT^1, \ldots) \rightarrow B(X, \ldots)\ldots$$

while the *successor* operation, since bune clauses are required, can be expressed by

$$A(X, \ldots, W_1 X T^1 W_2) \rightarrow B(XT^1, \ldots)\ldots$$

if the last argument of A is universal. The previous two clauses are both valid for normal form or non-normal form counters. For normal form counters, the variable W_1 can be omitted.

Let us consider a first example to illustrate these notions of basic operations on counters.

3.1.1. *MIX*

Let *MIX* be the language whose words are strings of a, b and c, in any order, that contain the same number of occurrences of each letter

$$MIX = \{w \in \{a,b,c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w)\}$$

where $\#_\sigma(w)$ denotes the number of occurrences of the symbol σ in the string w.

This language, with a very simple and intuitive characterization, is famous in formal language theory and has been used in well-known standard exercises. Yet, the place of *MIX* in the hierarchy of formal languages is a difficult problem, it may well be outside the class of indexed languages (ILs). In [12] the authors have shown that *MIX* can be defined by a variant of TAGs with local dominance and linear precedence, but very little is known about this class of grammars, except that, as TAGs and other MCS formalisms, they satisfy the constant growth property. Below, we show that *MIX* is an RCL.

Consider the following set of clauses

$$\begin{aligned}
&1\text{: } S(XYZ) && \rightarrow eqlen(X,Y)\ eqlen(X,Z)\ M(X,X,X,XYZ)\\
&2\text{: } M(XT^1,Y,Z,aW) && \rightarrow M(X,Y,Z,W)\\
&3\text{: } M(X,YT^1,Z,bW) && \rightarrow M(X,Y,Z,W)\\
&4\text{: } M(X,Y,ZT^1,cW) && \rightarrow M(X,Y,Z,W)\\
&5\text{: } M(\varepsilon,\varepsilon,\varepsilon,\varepsilon) && \rightarrow \varepsilon
\end{aligned}$$

In the main predicate M of arity four, the three leftmost arguments are counters, each of them counts (or more precisely counts down) the number of occurrences of a given letter in an input string: the first one counts the a's, the second one the b's, and the third one the c's. Any input string is examined from left to right and the rightmost argument holds the suffix of the input string not yet processed. At each step of the derivation process, the value of the counter for each letter must be the number of occurrences of that letter in the suffix part. These three counters (in normal form), are all initialized in clause #1 with the same value, which equals the third of the input string length. Due to the fact that the *eqlen* nonterminal is predefined, the first clause is instantiated if and only if the length of the input string is a multiple of three and, in that case, the corresponding instantiated clause is unique and is computed, at parse time, in constant time. In clauses #2, #3 or #4, if the leftmost letter of the unprocessed suffix is an a, b or c, this letter is stripped off that suffix and, if possible, the corresponding counter is

decreased by one, until, eventually, the three counters are null and the suffix is empty (clause #5). Each suffix of the input string is examined, once for each possible leading letter (clauses #2, #3 and #4) and once for emptiness (clause #5). Each of these four processings takes a constant amount of time but at most one succeeds. In that case, the corresponding counter is decremented by one (in constant time) and the whole process is iterated with a smaller suffix, unless this suffix is empty.

Thus, the *MIX* language is an RCL which can be recognized in linear time.

Of course, we can think of several generalizations of *MIX*. Below, we examine two of them, one in which the relation between the number of occurrences of each letter is not the equality and a second one in which the number of letters is not limited to three.

As a first generalization of *MIX*, consider the language

$$MIX_{xyz} = \{w \in \{a,b,c\}^* \mid \#_a(w) = kx, \#_b(w) = ky, \#_c(w) = kz, k \geqslant 0\}.$$

For example, to get the grammar of MIX_{213}, we simply have to change the first clause of *MIX* (which is in fact the language MIX_{111}) into

$$1' : S(X_1X_2YZ) \rightarrow eqlen(X_1X_2Y, Z)\ eqlen(X_1, X_2)\ eqlen(X_1, Y)$$
$$M(X_1X_2, X_1, X_1X_2Y, X_1X_2YZ)$$

In [13], we have shown that languages such as $\{a^p b^q c^r \$ w \mid w \in MIX_{pqr}\}$, in which the prefix of each word, before the dollar character, (dynamically) specifies the relative quantities of a, b and c's, are also RCLs.

The *MIX* language has been defined on the alphabet $T = \{a,b,c\}$. We can generalize *MIX* to any finite alphabet T and define the language MIX^T: if $T = \{a_1, \ldots, a_i, \ldots, a_{|T|}\}$, we have $MIX^T = \{w \in T^* \mid \#a_1(w) = \cdots = \#a_i(w) = \cdots = \#a_{|T|}(w)\}$ (remark that $MIX = MIX^{\{a,b,c\}}$). This language MIX^T is an RCL that can be defined by an RCG in which the arity of the nonterminal M is $|T| + 1$: each of its first $|T|$ arguments are counters devoted to a particular symbol of T. In that case, a typical M definition for the processing of the symbol a_i would be

$$M(X_1, \ldots, X_iT^1, \ldots, X_{|T|}, a_iW) \rightarrow M(X_1, \ldots, X_i, \ldots, X_{|T|}, W)$$

We can easily see that the corresponding parser still runs in linear time.

A more subtle generalization of *MIX* is the set $\bigcup_{U \subseteq T} MIX^U$. This language is still an RCL in which the arity of the nonterminal M is now $|T| + 2$: each of its first $|T|$ arguments are counters in normal form, the ith counter is devoted to a_i. This RCL can be defined by

$$1:\ S(T^0Y) \rightarrow M(T^0, \ldots, T^0, \ldots, T^0, Y, Y)$$

$$\ldots$$

2_i: $M(X_1,\ldots,X_i,\ldots,X_{|T|},$
$a_iY,X_iT^1Z) \rightarrow M(X_1,\ldots,X_iT^1,\ldots,X_{|T|},Y,X_iT^1Z)$

$\ldots$

3: $M(X_1,\ldots,X_i,\ldots,X_{|T|},$
$\varepsilon,W) \rightarrow A_{|T|}(X_1,\ldots,X_i,\ldots,X_{|T|})$

$4_{|T|}$: $A_{|T|}(X_1,\ldots,X_{|T|-1},$
$X_{|T|-1}) \rightarrow A_{|T|-1}(X_1,\ldots,X_{|T|-1})$

$4'_{|T|}$: $A_{|T|}(X_1,\ldots,T^1X_{|T|-1},\varepsilon) \rightarrow A_{|T|-1}(X_1,\ldots,T^1X_{|T|-1})$

$4''_{|T|}$: $A_{|T|}(X_1,\ldots,\varepsilon,T^1X_{|T|}) \rightarrow A_{|T|-1}(X_1,\ldots,T^1X_{|T|})$

$\ldots$

4_i: $A_i(X_1,\ldots,X_{i-1},X_{i-1}) \rightarrow A_{i-1}(X_1,\ldots,X_{i-1})$

$4'_i$: $A_i(X_1,\ldots,T^1X_{i-1},\varepsilon) \rightarrow A_{i-1}(X_1,\ldots,T^1X_{i-1})$

$4''_i$: $A_i(X_1,\ldots,\varepsilon,T^1X_i) \rightarrow A_{i-1}(X_1,\ldots,T^1X_i)$

$\ldots$

4_2: $A_2(X_1,X_1) \rightarrow \varepsilon$

$4'_2$: $A_2(T^1X_1,\varepsilon) \rightarrow \varepsilon$

$4''_2$: $A_2(\varepsilon,T^1X_2) \rightarrow \varepsilon$

The clause #1 initializes the $|T|$ counters of M to 0. The two rightmost arguments are both initialized to the initial range. The last one is a universal argument that stays unchanged throughout the M calls. Clause $\#2_i$ processes the letter $a_i \in T$. Note how the universal argument is used to maintain counters in normal form at 0. When clause #3 is instantiated, the value of each counter is the number of occurrences of the corresponding letter in the whole input string. Thus, we simply have to check that the values of each non-null counters are equal. This check is performed by the A_i-clauses. Each predicate A_i, $2 \leqslant i \leqslant |T|$ is of arity i and examines its two rightmost arguments, all the others, if any, are passed unchanged to A_{i-1}. Our checking condition is fulfilled either when the two last counters are identical (clause $\#4_i$) or when one is null (clauses $\#4'_i$ and $\#4''_i$).

Since clause #3 is reached in time linear with n while the check on the $|T|$ counters is performed in constant time (for a given grammar), the corresponding parser has a

linear time complexity. This result improves a result in [13] in which this language is recognized in quadratic time.

3.2. Addition, multiplication, ...

So far, we have seen how basic arithmetic operations can be performed on counters. The current section is dedicated to more complex operations together with some applications.

In the sequel, some nonterminal denotations will be surrounded by brackets such as [ ].

As an example, the result Z of the addition of two counters X and Y can be performed by the predicate call $[x+y=z](X,Y,Z)$, where $[x+y=z]$ is a nonterminal defined by

$$[x+y=z](X,Y,Z_1Z_2) \rightarrow eqlen(X,Z_1)\ eqlen(Y,Z_2)$$

while the result Z of the multiplication of two counters X and Y can be performed by the predicate call $[x*y=z](X,Y,Z)$, where $[x*y=z]$ is a nonterminal symbol defined by

$$[x*y=z](X,YT^1,Z_1Z_2) \rightarrow eqlen(X,Z_2)\ [x*y=z](X,Y,Z_1)$$
$$[x*y=z](X,\varepsilon,\varepsilon) \rightarrow \varepsilon$$

Of course, the two previous nonterminals do not really perform an addition or a multiplication since their 'result' is not an integer counter (predicates are boolean functions and thus could only play the role of boolean counters). In fact, they merely check, whether or not the value of their third counter is the sum or the product of the first two counters. To be more precise, when one wants to perform say an addition between two counters X and Y, one must guess the result in the third counter Z. The previous predicate definitions only check the validity of that guess.

Note that an addition takes constant time. For a multiplication, any given predicate call $[x*y=z](X,Y,Z)$ delivers its answer (yes or no) in a number of steps which is bounded by $2*(1+\min(|Y|,|Z|/|X|))$ (there are $1+\min(|Y|,|Z|/|X|)$ predicate calls and each predicate call is examined for instantiation by two exclusive clauses and each of them can be instantiated at most once). Thus its parse time is $\mathcal{O}(\min(|Y|,|Z|/|X|))$.

The predicate $[x+y=z]$ can also be used to perform the subtraction operation: for example $|Y|$, the value of the Y-counter is equal to $|Z|-|X|$. Similarly, $[x*y=z]$ can be used for a division: $|Y|=|Z|/|X|$, and the predicate call $[x*y=z](X,X,Z)$ 'calculates' the square of $|X|$ or the square root of $|Z|$.

3.3. Duplication

Duplication is one of the three classical phenomena, together with multiple agreements and cross agreements, that are not context-free and that any MCS formalism must be able to handle. The simplest duplication phenomenon is usually abstracted by

the 2-*copy language*, defined over some alphabet T by $\{ww \mid w \in T^*\}$. This language can be defined by a single clause

$$[w^2](X_1X_2) \to eq(X_1, X_2)$$

which is parsed in linear time.

Of course, the k-*copy language* $\{w^k \mid w \in T^*\}$, for any fixed integer $k \geqslant 2$, can be defined by a single clause

$$S(X_1X_2 \ldots X_k) \to \bigwedge_{i=2}^{k} eq(X_1, X_i)$$

which can also be parsed in linear time.

However, if we want a definition that works for any k, we can define a ternary predicate $[w^k]$ whose first argument binds to the string w, the second argument is a counter whose value is k, and the third argument binds to the whole string w^k. The corresponding $[w^k]$-clauses are much like the previous $[x * y = z]$-clauses, except that the substrings X and Z_2 must be equal instead of only having the same length.

$$[w^k](X, YT^1, Z_1Z_2) \to eq(X, Z_2)[w^k](X, Y, Z_1)$$
$$[w^k](X, \varepsilon, \varepsilon) \to \varepsilon$$

We can easily see that the parse time complexity of the call $[w^k](X, Y, Z)$ is $\mathcal{O}(\min(|X|*|Y|, |Z|))$: if the counter $|Y| = k$ is exhausted first, we have performed $|X|*|Y|$ letter comparisons of the suffix part of Z and if Z is exhausted first, all its constituent letters have been compared once. Thus, if such a call succeeds, it takes a time linear with $|Z|$.

Next we consider the *-*copy language* $\{w^* \mid w \in T^*\} = \bigcup_{k \geqslant 0}\{w^k \mid w \in T^*\}$. This ambiguous language can be defined by the following set of clauses:

$$S(XY) \to [w^*](X, Y)$$
$$[w^*](T^1X, Y) \to [w^+](T^1X, Y)$$
$$[w^*](\varepsilon, \varepsilon) \to \varepsilon$$
$$[w^+](X, YZ) \to eq(X, Z)[w^+](X, Y)$$
$$[w^+](X, \varepsilon) \to \varepsilon$$

The first clause guesses the prefix part $w = X$, and checks through the $[w^*](X, Y)$ calls that the suffix part Y has the form w^*. At worst, $[w^+](X, Y)$ gives its answer in time linear with $|Y|$: the predefined eq predicate, in the worst case, has to compare once each letter of an input string. The total quadratic behavior is due to the fact that the number of calls $[w^+](X, Y)$ in the first clause is linear with $n = 1 + |X| + |Y|$. Thus, the $*$-copy language is an RCL that can be parsed in quadratic time.

If $w \in T^*$, the $*$-copy language is an RCL, but this results still holds if w is more constrained and in particular if $w \in \mathscr{L}$, where $\mathscr{L}$ is an RCL. Assume that L is the

axiom of the RCG defining $\mathscr{L}$, the language $\{w^* \mid w \in \mathscr{L}\}$ is an RCL that can be defined by

$$S(XY) \rightarrow [w^*](X,Y)L(X)$$

Another generalization of duplication we may think of (among many others) is the $|w|$-*copy language* $\{w^{|w|} \mid w \in T^*\}$ in which a string w is duplicated $|w|$ times to form a word. This language may be seen as a generalized case of the k-copy language for which we have $k = |w|$. Thus this language can be defined by the clause

$$[w^{|w|}](XY) \rightarrow [w^k](X,X,XY)$$

in which the two first arguments of the predicate call $[w^k](X,X,XY)$ take the same value: the first one stands for the string w, while the second one is a counter whose value is initialized with $|w|$.

Since the parse time complexity of the call $[w^k](X,X,XY)$ is $\min(|X|^2, |XY|)$, the parse time complexity of $[w^{|w|}]$, for an input string of length n is

$$\sum_{|X|=0}^{n} \min(|X|^2, n)$$

that is $\mathcal{O}(\sum_{|X|=0}^{\sqrt{n}} |X|^2 + \sum_{\sqrt{n}}^{n} n) = \mathcal{O}(n\sqrt{n}) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$. We shall see in the next section how this quadratic behavior can be improved.

3.4. Squares and square roots

We have already mentioned how the nonterminal $[x * y = z]$ defined in Section 3.2 can be used to calculate a square or a square root. For example, the *square language* $\{w \in T^* \mid |w| = p^2, p \geqslant 0\}$, can be defined by

$$[x^2](XY) \rightarrow [x * y = z](X,X,XY)$$

This clause cuts, in all possible ways, the input text w, $|w| = n$ into a prefix part denoted by X, $0 \leqslant |X| \leqslant n$ and a suffix part denoted by Y and the predicate call $[x * y = z](X,X,XY)$ checks whether or not we have $|X|^2 = n$. We already know that, for each call $[x * y = z](X,X,XY)$, the predicate $[x * y = z]$ delivers its answer (true or false) in $1 + \min(|X|, |Z|/|X|)$ steps. Therefore, the total number of steps τ performed by the predicate call $[x * y = z](X,X,XY)$, for all X values, is $\sum_{i=0}^{n}(1 + \min(i, n/i))$. Since $\min(i, n/i)$ is equal to i if $i \leqslant \sqrt{n}$ and is equal to n/i if $i \geqslant \sqrt{n}$, we get $\tau \leqslant (n+1) + \sum_{i=0}^{\sqrt{n}} i + \sum_{i=\sqrt{n}}^{n} n/i$ whose asymptotic behavior is $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$. Therefore, the square language can be parsed in $\mathcal{O}(n \log n)$ time.

For the square language, we can define a more efficient grammar that is based upon the following elementary property: the pth perfect square (i.e., p^2) is the sum of the

first p odd integers. The following set of RCG clauses uses this property.

$$
\begin{aligned}
&1\colon [x^2](\varepsilon) && \to \varepsilon\\
&2\colon [x^2](T^1X) && \to [\sqrt{x}](T^1, T^1, X, T^1X)\\
&3\colon [\sqrt{x}](X, P, YT^2Z, PT^1W) && \to eqlen(X, Y)[\sqrt{x}](YT^2, PT^1, Z, PT^1W)\\
&4\colon [\sqrt{x}](X, P, \varepsilon, W) && \to \varepsilon
\end{aligned}
$$

The empty word is processed by the first clause while nonempty input strings trigger the second clause. The four arguments of each predicate call of the form $[\sqrt{x}](X, P, Y, W)$ have the following meaning: P is a counter whose value is i (we process the ith odd integer), X is a counter whose value is $2i-1$, the ith odd integer, W is a universal argument that binds to the whole input string while Y is the suffix of W such that $UY = W$ and $|U| = i^2$. When eventually the third argument becomes empty in clause #4, the elementary property shows that the value of the second argument is $\sqrt{|W|}$. Note how in clause #3, the odd integer $|Y| + 2|$ next to $|X|$ is computed and how the P-counter is incremented, using the universal argument. If we look at the parse time complexity of the previous grammar, we can easily see that the length of any derivation (complete or not), for an input string of length n is $1+\sqrt{n}$. Since for a given instantiated predicate there is at most one instantiated clause whose (instantiated) arguments are computed in constant time, the total parse time for this grammar is $\mathcal{O}(\sqrt{n})$.

In other words, there exist some nontrivial RCLs (the square language is not MCS since it does not possess the constant growth property), that can be parsed in sublinear time!

Note that the empty right-hand side of clause #4 can be replaced by any conjunction of predicate calls that wants to exploit the value $p = \sqrt{|W|}$ of the P-counter and the whole input string W. For example, if we replace clause #4 by

$$4'\colon [\sqrt{x}](X, P, \varepsilon, W) \to [w^k](P, P, W)$$

the resulting grammar now (re)defines the language $\{w^{|w|} \mid w \in T^*\}$. The first part of this grammar checks that the length of an input string x is a perfect square, say $n = p^2$, in time $\mathcal{O}(p)$, if it is the case, the call $[w^k](P, P, W)$ checks that x has the form $x = w^{|w|}$, $|w| = p$, in time linear with n. Thus, the $|w|$-copy language $\{w^{|w|} \mid w \in T^*\}$ can be parsed in linear time.

A variant of the $|w|$-copy language has been introduced in [14]. This language, defined by $nonIL_1 = \{(\$w)^{|w|} \mid w \in T^* \wedge \$ \notin T\}$, is interesting because it has been proved not to be an IL. First note that this language is a simplified version of the $|w|$-copy language in the sense that the w slices have not to be discovered but are explicitly marked by \$ signs. This language will help us to study the relationship between ILs and RCLs. We know that the context-sensitive languages (CS) can be accepted in exponential time, and that ILs are a proper subfamily of CS. However, it is unknown whether ILs contain languages that require at least exponential time (in [15] it is shown that the membership problem for IL is NP-complete). Since RCLs can

be accepted in polynomial time, if we assumed that there are some ILs that cannot be accepted in less than exponential time, this would show that these ILs are not RCLs. But are there some RCLs that are not ILs? The answer would be positive, if we admit that the *MIX* language (see Section 3.1.1), as conjectured, is not an IL. A definitive positive answer is proved below by showing that $nonIL_1$ is an RCL. It is defined by the following set of clauses:

$$
\begin{aligned}
&1\text{: } nonIL_1(\$X) && \rightarrow A(\$X, X)\\
&2\text{: } nonIL_1(\varepsilon) && \rightarrow \varepsilon\\
&3\text{: } A(X, T^1 Y) && \rightarrow \overline{[\$](T^1)} A(X, Y)\\
&4\text{: } A(\$X\$Y, \$Y) && \rightarrow B(\$X, X, \$X\$Y)\\
&5\text{: } A(\$X, \varepsilon) && \rightarrow B(\$X, X, \$X)\\
&6\text{: } [\$](\$) && \rightarrow \varepsilon\\
&7\text{: } B(X, YT^1, Z_1Z_2) && \rightarrow eq(X, Z_2)B(X, Y, Z_1)\\
&8\text{: } B(X, \varepsilon, \varepsilon) && \rightarrow \varepsilon
\end{aligned}
$$

The purpose of the A-clauses is to compute the leftmost pattern $\$w$ and then to initialize a counter, the second argument of the B predicate, with the value $|w|$. This counter is used by the B-clauses to check that the pattern $\$w$, held in the first argument, occurs $|w|$ times in the input string.

We can easily verify that this language is parsed in linear time. Thus we have shown that, rather surprisingly, some non-ILs can be parsed (very) efficiently. In Sections 3.5 and 3.6, we shall see other non-ILs, that are RCLs nevertheless.

3.5. Exponential and logarithm

If we reconsider the previous result on sublinear parse times, their existence seems strange since each symbol in any input string has to be read at least once. However, if we recall Section 2.4, it does make sense since the time taken by the scanner phase to process an input text w is not taken into account within our parse time complexities.

Thus, under this assumption, the regular language T^* itself can be defined by a single clause

$$[T^*](X) \rightarrow \varepsilon$$

whose parse time is $\mathcal{O}(1)$.

For a more convincing argumentation, let us consider the *exponential language* $\{w \in T^* \mid |w| = 2^l, l \geqslant 0\}$. This language does not possess the constant growth property, and thus is not MCS.

However, it can be described by

$$[2^x](XY) \rightarrow eqlen(X,Y)[2^x](X)$$
$$[2^x](T^1) \rightarrow \varepsilon$$

The predefined predicate *eqlen* constrains the sizes of X and Y in the first clause to be equal. Thus, each input string is cut into two parts of equal length, and, at each call, only the prefix part is processed. Therefore, for each input string of length n such that $2^{l-1} < n \leqslant 2^l$, $[2^x]$ gives its answer in at most l steps. And thus, we get an $\mathcal{O}(\log n)$ parse time. This gives a logarithmic parse time for a non-MCS language.

The variant $nonIL_2 = \{l^m(cl^m)^{2^m-1} \mid m \geqslant 1\}$ has been proposed in [16] where it is shown that this language is not an IL. However, it is an RCL that can be defined by the clauses

$$1:\ nonIL_2(W) \rightarrow A(W,W)$$
$$2:\ A(W,lX) \rightarrow A(W,X)$$
$$3:\ A(XcY,cY) \rightarrow B(X,X,XcY)$$
$$4:\ B(XT^1,Y,Z_1cZ_2) \rightarrow eq(Z_1,Z_2)B(X,Y,Z_1)$$
$$5:\ B(\varepsilon,Y,Y) \rightarrow \varepsilon$$

The purpose of the A-clauses is to compute the value m which is used to initialize the first argument of B as a counter, the two rightmost arguments are, respectively, the prefix l^m and the whole input string w. If w is a word, clause #4 is executed exactly m times. At each step, the remaining prefix of w is cut according to the pattern Z_1cZ_2 in which substrings Z_1 and Z_2 are equal. In clause #5, when the counter is null, we check that the remaining prefix of w is the prefix l^m. Since each character in the suffix part Z_2 is checked once by the *eq* predicate in clause #4, the total parse time is linear in $|w|$.

RCGs can also be used to compute logarithms. As an example, the following predicate $[\log_2 x]$ calculates in its first counter the smallest integer greater than or equal to the base 2 logarithmic value of its second counter. We assume that, initially, the value of the first counter is 0 and the value of the second counter is n, $n \geqslant 1$. This calculation succeeds for all integers such that $n \geqslant 1$. Both counters are in normal form.

$$1:\ [\log_2 x](X,T^1) \rightarrow \varepsilon$$
$$2:\ [\log_2 x](X,XT^1W_1W_2) \rightarrow eqlen(XT^1W_1,W_2)[\log_2 x](XT^1,XT^1W_1)$$
$$3:\ [\log_2 x](X,XT^1W_1T_1^1W_2) \rightarrow eqlen(XT^1W_1,W_2)[\log_2 x](XT^1,XT^1W_1T_1^1)$$

If the second counter is an even number say $2k$ (clause #2), the first counter is incremented by one, and the second counter is divided by two and thus takes the value k. For an odd number, say $2k+1$, if $k=0$ the computation stops at clause #1 and the value X of the first counter is $\lceil \log_2 n \rceil$). It is not difficult to see that at each derivation step, we have to examine three clauses and that a single one is deterministically

selected in constant time. Moreover, that single clause has a single instantiation for a given instantiated left-hand side $[\log_2 x]$-predicate. Thus, this computation can be performed in $\mathcal{O}(\log n)$ time.

3.6. Factorial

In [14], it has been shown that the *factorial language* $\{w \in T^* \mid |w| = l!, l \geqslant 1\}$ is not an IL. However, this language is an RCL defined by

$$
\begin{aligned}
&1\text{: } [x!](T^1) && \to \varepsilon\\
&2\text{: } [x!](T^2Y) && \to [x! = y](T^2, T^2Y)\\
&3\text{: } [x! = y](X,X) && \to \varepsilon\\
&4\text{: } [x! = y](X, T^0XT^1Y) && \to [z/x = y](X, T^0, T^0XT^1Y, T^0XT^1Y)\\
&5\text{: } [z/x = y](X, Y, Z_1Z_2, YT^1W) && \to eqlen(X, Z_2)\\
&\qquad\qquad && \quad\ [z/x = y](X, YT^1, Z_1, YT^1W)\\
&6\text{: } [z/x = y](X, Y, \varepsilon, XT^1W) && \to [x!](XT^1, Y)
\end{aligned}
$$

Clauses #1 captures 1! while $l!$, $l \geqslant 2$ is processed by clause #2. In the right-hand side of clause #2, the first argument of the predicate $[x! = y]$ is a normal form counter initialized with 2 and which will eventually contain the value l; its second argument is a normal form counter initialized with $|w| = n \geqslant 2$, if w is the input string. The idea is to iteratively divide the second counter by the first one, the values of these counters being for the dividend the successive quotient of these divisions, starting with n, and for the divisor the values $2, 3, 4$, etc. This process succeeds if and only if dividend and divisor become equal (clause #3). The division operation is performed by the nonterminal $[z/x = y]$. A $[z/x = y](X, Y, Z, W)$ call calculates in its Y counter the quotient of Z by X. Note that W is universal. Initially, the value of the quotient counter Y is null, the value of the dividend counter Z is n, the value of the divisor counter X is 2, and the value of the universal argument is n. At the end (clause #6), the division succeeds when the value of the dividend counter Z is null. Then, the $[x!]$ predicate is called with the next divisor value and the quotient value. Of course, this quotient value must be greater than or equal to the new divisor value (resp. clause #4 and clause #3).

The computation of the quotient value by a $[z/x = y](X, Y, Z, W)$ call is performed in $|Z|/|X|$ steps. Thus, the first quotient value ($|Z| = l!$ and $|X| = 2$), takes $l!/2$ steps. The next ones take $l!/3!, \ldots, l!/i!, \ldots, l!/l!$ steps. Any complete derivation is performed in $\mathcal{O}(l! \sum_{i=2}^{i=l} 1/i!)$ steps. At each step, the choice of the clause to instantiate as well as the corresponding instantiation are performed in constant time. Since $\sum_{i=2}^{i=l} 1/i! < 1$, the total parse time of the factorial language is linear in $n = l!$.

We conclude with two non-trivial examples that both show the power of RCGs and illustrate the use of counters.

3.7. Prime numbers

The first example deals with prime numbers. Given some vocabulary T, let a *prime string* be a string in T^* whose length is a prime number. For some T, the set of all prime strings is called a *prime language*. Any prime language is an RCL that can be defined by the following RCG clauses:

$$
\begin{aligned}
&1\text{:}\ \ Prime(X) && \rightarrow \overline{NotPrime(X)}\\
&2\text{:}\ \ NotPrime(\varepsilon) && \rightarrow \varepsilon\\
&2'\text{:}\ \ NotPrime(T^1) && \rightarrow \varepsilon\\
&3\text{:}\ \ NotPrime(T^2XYZ) && \rightarrow eqlen(T^2X,Y)[x*y \leqslant z](Y,Y,T^2XYZ)\\
& && \quad\ [x*?=y](Y,Z)\\
&4\text{:}\ \ [x*y \leqslant z](X,\varepsilon,Z) && \rightarrow \varepsilon\\
&5\text{:}\ \ [x*y \leqslant z](X,YT^1,Z_1Z_2) && \rightarrow eqlen(X,Z_2)[x*y \leqslant z](X,Y,Z_1)\\
&6\text{:}\ \ [x*?=y](X,\varepsilon) && \rightarrow \varepsilon\\
&7\text{:}\ \ [x*?=y](X,Y_1Y_2) && \rightarrow eqlen(X,Y_2)[x*?=y](X,Y_1)
\end{aligned}
$$

We choose to define the set of prime strings as the complement of the set of nonprime strings (see clause #1). Though it is clear that negative predicate calls directly bring into RCG framework the property of closure under complement, and much more flexibility in the design of some practical grammars (even in NL processing), it has not yet been shown that NRCGs extend the formal power of PRCGs. However, we were not able to design a PRCG for the prime language. Clause #2 and #2′ indicate that neither 0 nor 1 are prime numbers. A strictly positive integer number n is not a prime number if and only if there are two integers x and k, $1<x\leqslant k<n$ such that $n=kx$. More precisely, we know that a tighter upper bound for x and k is the square root of n (i.e., $2\leqslant x\leqslant k\leqslant\sqrt{n}$). Since 2 and 3 are prime numbers, a number $n\geqslant 4$ is not a prime number if and only if we have the following conditions: there are three numbers x, y and z such that $x+y+z=n$, $2\leqslant x$, $x=y$, $y^2\leqslant n$ and z is a multiple of y ($\exists h\geqslant 0 : z-x-y=hy$). Thus, clause #3 may be read as: n is not a prime number if and only if the input range of length n may be decomposed into four consecutive ranges: T^2 (of size 2) and X, Y and Z such that $|X|\geqslant 0$, $2+|X|=|Y|$, $|Y|^2\leqslant n$ (checked by the predicate $[x*y\leqslant z]$) and $|Z|$ is a multiple of $|Y|$ (checked by $[x*?=y]$). Predicate $[x*y\leqslant z]$ is defined by clauses #4 and #5. These clauses are variants of the $[x*y=z]$-clauses defined in Section 3.2 in which the halting condition has been modified. Predicate $[x*?=y]$, defined by clauses #6 and #7, is another variant of predicate $[x*y=z]$ in which the multiplier value is unspecified.

Now, let us consider its parse time complexity. For the $[x*?=y]$-clauses, we can easily see that a test to determine whether two ranges $|X|$ and $|Y|$ are such that $|Y|$ is a multiple of $|X|$ is performed in exactly $|Y|/|X|+1$ calls and thus takes $\mathcal{O}(|Y|/|X|)$ time. Analogously, we can see that the time for any predicate call $[x*y\leqslant z](X,Y,Z)$ is

$\mathcal{O}(|Y|)$ when it succeeds or $\mathcal{O}(|Z|/|X|)$ when it fails. In clause #3, the only unknown is the upper bound of X since the lower bound of T^2 is 0, the upper bound of Z is n, $|T^2|=2$, $|Y|=|T^2|+|X|$ and $|Z|=n-(|T^2|+|X|+|Y|)$. This upper bound takes at most $n/2$ different values. The complexity due to the execution of the nonterminal $[x*y\leqslant z]$ is the sum of two terms: one when its execution succeeds and the other one when its execution fails. We know that the execution succeeds if and only if $|X|\leqslant\sqrt{n}$. Therefore, the execution time for the predicate $[x*y\leqslant z]$ is $\sum_{x=2}^{\sqrt{n}} x+\sum_{x=\sqrt{n}}^{n/2} n/x$ whose limit is $\mathcal{O}(n)+\mathcal{O}(n\log n)=\mathcal{O}(n\ \log\ n)$. The execution time for the predicate $[x*?=y]$ is $\sum_{x=2}^{n/2} n/x$ whose limit is $\mathcal{O}(n\ \log\ n)$.

If we assume that in an instantiated clause its right-hand side predicate calls are executed from left to right, the upper bound of the $\sum$ for predicate $[x*?=y]$ is $\sqrt{n}$ instead of $n/2$ since $[x*?=y]$ is only executed when $[x*y\leqslant z]$ succeeds. However, this consideration does not improve the $\mathcal{O}(n\log n)$ limit. Many other cosmetic changes may produce very effective improvements, but none succeeds to beat this $\mathcal{O}(n\log n)$ boundary.

The parse time complexity of the prime language is $\mathcal{O}(n\ \log\ n)$.

Since RCGs are closed under intersection, the language $\{w^{|w|}\mid w\in T^*\}$ where $|w|$ is a prime number can be simply defined in changing clause $4'$ in Section 3.4 by

$$4'':\ [\sqrt{x}](X,P,\varepsilon,W)\rightarrow[w^k](P,P,W)Prime(P)$$

and it is not difficult to see that its time complexity stays linear in $n=|w^{|w|}|$.

3.8. Numbers in base p

The purpose of this section is to show that single counters are not necessarily constrained to be associated to a single argument (i.e., the value of a single counter may be coded on several arguments).

Consider the language $L_k=\{a^pb^q\mid p\geqslant 2, q<p^k\}$, $k>0$, where the number of a's and the number of b's follow the specified constraint. One way to define this language is to represent q in base p on k digits. Of course, a string is a sentence if and only if this representation succeeds. As an example, we discuss the case $k=4$.

Consider an input string $w=a^pb^q$ and assume that the initial range $\langle 0..p+q\rangle$ is denoted by aPT^0Q such that each variable P, T^0 and Q, respectively, binds to the ranges $\langle 1..p\rangle$, $\langle p..p\rangle$ and $\langle p..p+q\rangle$. The predicate *Base*, which performs the transformation of q in base p on four digits, is of arity six, the four first counters are the four digits of the representation of q in base p while the two last counters hold, respectively, $p-1$ and the part of q that has not yet been represented in four digits. Thus, the initial predicate call has the form $Base(T^0,T^0,T^0,T^0,P,Q)$. The P-counter holds the value $p-1$ and stays unchanged throughout the various *Base* calls. The idea of the following *Base*-clauses is to strip Q from right to left, letter by letter, until completion, while incrementing its base p representation. Of course, the only difficulty is the management of the carry: a carry is produced (and propagated) each time the

unit digit of the representation (the fourth counter), holds the value $p-1$ (i.e., is equal to the P-counter).

$$
\begin{aligned}
&Base(D_3, D_2, D_1, D_0, PT^1D_0, QT_0^1) \rightarrow Base(D_3, D_2, D_1, T^1D_0, PT^1D_0, Q)\\
&Base(D_3, D_2, D_1, PT^1D_1,\\
&\qquad PT^1D_1T^0, QT_0^1) \rightarrow Base(D_3, D_2, T^1D_1, T^0, PT^1D_1, Q)\\
&Base(D_3, D_2, PT^1D_2, PT^1D_2,\\
&\qquad PT^1D_2T^0, QT_0^1) \rightarrow Base(D_3, T^1D_2, T^0, T^0, PT^1D_2, Q)\\
&Base(D_3, PT^1D_3, PT^1D_3, PT^1D_3,\\
&\qquad PT^1D_3T^0, QT_0^1) \rightarrow Base(T^1D_3, T^0, T^0, T^0, PT^1D_3, Q)\\
&Base(D_3, D_2, D_1, D_0, P, \varepsilon) \rightarrow \varepsilon
\end{aligned}
$$

The first clause handles the case where the value of the unit counter is strictly less than $p-1$. The three middle clauses are, respectively, dedicated to the cases where the values of the rightmost one, two, or three counters of the base p representation are $p-1$. The last clause eventually stops the conversion process.

We can easily see that this transformation is performed in time linear with q. Moreover, when Q is exhausted in the last clause, we have, in the four first counters, the canonical representation of q in base p on four digits.

Using this multiple argument representation, for an input string of length n, a counter represented on k arguments can take any value v such that $0 \leqslant n^k$. Thus, the initial hypothesis that forces a counter to be bounded by the length of the current input string can be partly released.

4. Conclusion

In [8], we have introduced RCGs as a promising syntactic formalism. RCGs are powerful: they extend MCS formalisms and we have shown in [13] how this extra power can be used in NL processing to express Chinese numbers or scrambling, phenomena which are not MCS. In spite of their power, RCGs stay computationally tractable. Their associated parsers work in time polynomial with the size of the input string and in time linear with the size of the grammar. Moreover, in a given grammar, only complicated (many arguments, many variables) clauses produce higher parse times whereas simpler clauses induce lower times. As shown in [7], even the subclass of RCGs with a single argument, is already a powerful extension of CFGs. This subclass can be parsed in cubic time and contains both the intersection and the complement of CFLs.

For any given input string, the output of an RCG parser, which consists of an exponential or even unbounded number of derived trees, can always be represented into a compact structure, the shared forest, which is a CFG of polynomial size and from which each individual derived tree can be extracted in time linear in its own size.

As CFGs, RCGs may themselves be considered as a syntactic backbone upon which other formalisms such as Herbrand's domain or feature structures can be grafted.

And lastly, we have seen that RCGs are modular. This allows to imagine libraries of linguistic modules in which any language designer can pick up at will when he wants to specify such and such phenomena.

All these properties seem to advocate that RCGs might well have the right level of formal power needed in NL processing. However, the practical capability of RCGs to define real NL still has to be evaluated.

Alternatively, RCGs may be used as an intermediate high level implementation mechanism. The idea is to take a less powerful formalism and to translate this formalism into an equivalent RCG which is transformed in turn into an RCG parser. This idea has been investigated in [17] where we have shown that unrestricted TAGs and set-local multi-component TAGs can be translated into equivalent PRCGs. Moreover, these transformations do not induce any over-cost. For example, we get the classical $\mathcal{O}(n^6)$ parse time for TAGs.

The contribution of this paper is to show that some arithmetics can be performed within the RCG formalism and that RCGs are not so bad at sums. During this study, we have noticed that some of these arithmetic operations can be performed efficiently (for example the test that the length n of a string is a prime number takes $\mathcal{O}(n \log n)$ time). And, more surprisingly, we have identified operations like square root or logarithm that take sublinear times. We have also shown that some non-ILs can be parsed in linear time.

References

[1] A.V. Groenink, SURFACE WITHOUT STRUCTURE word order and tractability issues in natural language analysis, Ph.D. Thesis, Utrecht University, The Netherlands, 1997, 250pp.

[2] W.C. Rounds, Lfp: a logic for linguistic descriptions and an analysis of its complexity, ACL Comput. Linguistics 14 (4) (1988) 1–9.

[3] P. Boullier, Proposal for a natural language processing syntactic backbone, Research Report 3342, INRIA-Rocquencourt, France, January 1998, 41pp, at `http://www.inria.fr/RRRT/RR-3342.html`.

[4] D. Weir, Characterizing mildly context-sensitive grammar formalisms, Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA, 1988.

[5] K. Vijay-Shanker, D. Weir, A. K. Joshi, Characterizing structural descriptions produced by various grammatical formalisms, in: Proc. 25th Meeting of the Association for Computational Linguistics (ACL'87), Stanford University, CA, 1987, pp. 104–111.

[6] B. Lang, Recognition can be harder than parsing, Comput. Intell. 10 (4) (1994) 486–494.

[7] P. Boullier, A cubic time extension of context-free grammars, in: 6th Meeting on Mathematics of Language (MOL6), University of Central Florida, Orlando, FL, USA, 1999, pp. 37–50, see also Research Report No. 3611 at `http://www.inria.fr/RRRT/RR-3611.html`, INRIA-Rocquencourt, France, January 1999, 28pp.

[8] P. Boullier, Range concatenation grammars, in: Proc. 6th Int. Workshop on Parsing Technologies (IWPT 2000), Trento, Italy, 2000, pp. 53–64.

[9] A.K. Joshi, An introduction to tree adjoining grammars, in: A. Manaster-Ramer (Ed.), Mathematics of Language, John Benjamins, Amsterdam, 1987, pp. 87–114.

[10] R.L. Graham, D.E. Knuth, O. Patashnik, Concrete Mathematics — A Foundations for Computer Science, 2nd Ed., Addison-Wesley, Reading, MA, 1994.

[11] A.V. Aho, Indexed grammars — an extension of context-free grammars, J. ACM 15 (1968) 647–671.

[12] A.K. Joshi, K. Vijay-Shanker, D. Weir, The convergence of mildly context-sensitive grammatical formalisms, in: P. Sells, S. Shieber, T. Wasow (Eds.), Foundational Issues in Natural Language Processing, MIT Press, Cambridge, MA, 1991.

[13] P. Boullier, Chinese numbers, mix, scrambling, and range concatenation grammars, in: Proc. 9th Conf. of the European Chapter of the Association for Computational Linguistics (EACL'99), Bergen, Norway, 1999, pp. 53–60, see also Research Report No. 3614 at `http://www.inria.fr/RRRT/RR-3614.html`, INRIA-Rocquencourt, France, January 1999, 14pp.

[14] T. Hayashi, On derivation trees of indexed grammars — an extension of the *uvwxy*-theorem — , Report, Research institute of mathematical sciences, Kyoto University, 1973.

[15] W.C. Rounds, Complexity of recognition in intermediate-level languages, in: Proc. 14th Ann. IEEE Symp. Switching and Automata Theory, 1973, pp. 145–158.

[16] M.J. Fischer, Grammars with macro-like productions, in: Proc. 9th Ann. IEEE Symp. on Switching and Automata Theory, 1968, pp. 131–142.

[17] P. Boullier, On tag parsing and on multicomponent tag parsing, in: 6éme Conf. Ann. sur le Traitement Automatique des Langues Naturelles (TALN'99), Cargése, Corse, France, 1999, pp. 321–326, see also Research Report No. 3668 at `http://www.inria.fr/RRRT/RR-3668.html`, INRIA-Rocquencourt, France, April 1999, 39pp.